Introduction to SQL

Class 15

Course Overview

- Introduction to SQL
  - Databases, Tables
  - Classification of SQL – DDL, DML, DCL, TCL
    - DDL – CREATE, ALTER, DROP
    - DML – SELECT, INSERT, UPDATE, DELETE
    - DCL – GRANT, REVOKE
    - TCL – COMMIT, ROLLBACK, SAVEPOINT
  - Data types, Operators
  - Keys – Primary, Foreign, Composite, Unique, Alternate
  - Integrity Constraints – Domain Integrity Constrains, Entity Integrity Constraints, Referential Integrity Constraints
  - Joins – Outer Joins, Left Outer Joins, Right Outer Joins, Inner Joins.
  - Queries, Subqueries, Functions, Flow Control (IF, CASE, WHILE, REPEAT, LOOP), ,Stored functions ,Stored Procedures
  - Views
  - Indexes, Cursors, Triggers, Events
  - Concurrency and locking (Implicit locks, explicit locks, row level locks, table level locks, database level locks)
  - Tuning SQL queries and optimizing performance
  - SQL Databases vs NoSQL Databases
  - ACID, CAP
  - How SQL databases internally works

TRANSACTIONS

- A transaction is a sequential group of database manipulation operations, which is performed as if it were one single work unit
- In other words, a transaction will never be complete unless each individual operation within the group is successful
- If any operation within the transaction fails, the entire transaction will fail.

Real life examples of transactions

- Banking operations: Transactions can be used to ensure that banking operations, such as transferring funds from one account to another, are executed atomically and in a consistent manner.
- Online shopping: Transactions can be used to ensure that a customer's order is properly recorded in the database, including updating the inventory and creating an order record. This way, even if there is a failure in the middle of the process, the transaction can be rolled back to avoid partial updates or inconsistencies in the database.
- Stock market trading: Transactions can be used to ensure that a stock trade is executed atomically and in a consistent manner. For example, if a trade involves buying and selling stocks, the transaction ensures that either both operations succeed or both fail, avoiding partial updates or inconsistencies in the database.
- Flight booking: Transactions can be used to ensure that a flight booking is properly recorded in the database, including updating the available seats and creating a booking record. This way, even if there is a failure in the middle of the process, the transaction can be rolled back to avoid partial updates or inconsistencies in the database.
- Health care management: Transactions can be used to ensure that patient information is recorded atomically and in a consistent manner, avoiding partial updates or inconsistencies in the database.

Properties of transactions

- Transactions have the following four standard properties, usually referred to by the acronym **ACID**
- **Atomicity** – This ensures that all operations within the work unit are completed successfully; otherwise, the transaction is aborted at the point of failure and previous operations are rolled back to their former state.
- **Consistency** – This ensures that the database properly changes states upon a successfully committed transaction.
- **Isolation** – This enables transactions to operate independently on and transparent to each other.
- **Durability** – This ensures that the result or effect of a committed transaction persists in case of a system failure

Atomicity

- This property ensures that all statements or operations within the transaction unit must be executed successfully. Otherwise, if any operation is failed, the whole transaction will be aborted, and it goes rolled back into their previous state. It includes features:
  - COMMIT statement.
  - ROLLBACK statement.
  - Auto-commit setting.
  - Operational data from the INFORMATION_SCHEMA tables.

Consistency

- This property ensures that the database changes state only when a transaction will be committed successfully. It is also responsible for protecting data from crashes. It includes features:
  - InnoDB doublewrite buffer.
  - InnoDB crash recovery.

Isolation

- This property guarantees that each operation in the transaction unit operated independently. It also ensures that statements are transparent to each other. It includes features:
  - SET ISOLATION LEVEL statement.
  - Auto-commit setting.
  - The low-level details of InnoDB locking.

Durability

- This property guarantees that the result of committed transactions persists permanently even if the system crashes or failed. It includes features:
  - Write buffer in a storage device.
  - Battery-backed cache in a storage device.
  - Configuration option innodb_file_per_table.
  - Configuration option innodb_flush_log_at_trx_commit.
  - Configuration option sync_binlog.

COMMIT,ROLLBACK,SAVEPOINT

- In MySQL, the transactions begin with the statement **BEGIN WORK** and end with either a **COMMIT** or a **ROLLBACK** statement
- The SQL commands between the beginning and ending statements form the bulk of the transaction
- **COMMIT** - When a successful transaction is completed, the COMMIT command should be issued so that the changes to all involved tables will take effect.
- **ROLLBACK** - If a failure occurs, a ROLLBACK command should be issued to return every table referenced in the transaction to its previous state.

AUTOCOMMIT

- You can control the behavior of a transaction by setting session variable called AUTOCOMMIT
- If AUTOCOMMIT is set to 1 (the default), then each SQL statement (within a transaction or not) is considered a complete transaction and committed by default when it finishes
- When AUTOCOMMIT is set to 0, by issuing the **SET AUTOCOMMIT = 0** command, the subsequent series of statements acts like a transaction and no activities are committed until an explicit COMMIT statement is issued.

A Generic Example on Transaction

- Begin transaction by issuing the SQL command BEGIN WORK.
- Issue one or more SQL commands like SELECT, INSERT, UPDATE or DELETE.
- Check if there is no error and everything is according to your requirement.
- If there is any error, then issue a ROLLBACK command, otherwise issue a COMMIT command.

COMMIT Example

```
-- 1. Start a new transaction
START TRANSACTION;
-- 2. Get the highest income
SELECT @income:= MAX(income) FROM employees;
-- 3. Insert a new record into the employee table
INSERT INTO employees(emp_id, emp_name, emp_age, city, income)
VALUES (111, 'Alexander', 45, 'California', 70000);
-- 4. Insert a new record into the order table
INSERT INTO Orders(order_id, prod_name, order_num, order_date)
VALUES (6, 'Printer', 5654, '2020-01-10');
-- 5. Commit changes
COMMIT;
```

ROLLBACK Example

```
-- 1. Start a new transaction
START TRANSACTION;
-- 2. Delete data from the order table
DELETE FROM Orders;
-- 3. Rollback changes
ROLLBACK;
-- 4. Verify the records in the first session
SELECT * FROM Orders;
```

Statements that cannot be a rollback in using MySQL Transaction.

- MySQL Transaction cannot be able to roll back all statements. For example, these statements include DDL (Data Definition Language) commands such as CREATE, ALTER, or DROP database as well as CREATE, UPDATE, or DROP tables or stored routines. We have to make sure that when we design our transaction, these statements do not include.

SAVEPOINT, ROLLBACK TO SAVEPOINT, RELEASE SAVEPOINT

- The SAVEPOINT statement creates a special mark with the name of the identifier inside a transaction.
- It allows all statements that are executed after savepoint to be rolled back. So that the transaction restores to the previous state it was in at the point of the savepoint
- The ROLLBACK TO SAVEPOINT statement allows us to rolls back all transactions to the given savepoint was established without aborting the transaction.
- The RELEASE SAVEPOINT statement destroys the named savepoint from the current transaction without undoing the effects of queries executed after the savepoint was established.

SAVEPOINT savepoint_name

ROLLBACK TO [SAVEPOINT] savepoint_name

RELEASE SAVEPOINT savepoint_name

ROLLBACK TO SAVEPOINT example

```
START TRANSACTION;
SELECT * FROM Orders;
INSERT INTO Orders(order_id, prod_name, order_num, order_date)
VALUES (6, 'Printer', 5654, '2020-01-10');
SAVEPOINT my_savepoint;
INSERT INTO Orders(order_id, prod_name, order_num, order_date)
VALUES (7, 'Ink', 5894, '2020-03-10');
ROLLBACK TO SAVEPOINT my_savepoint;
INSERT INTO Orders(order_id, prod_name, order_num, order_date)
VALUES (8, 'Speaker', 6065, '2020-02-18');
COMMIT;
```

RELEASE SAVEPOINT

```
START TRANSACTION;
INSERT INTO Orders(order_id, prod_name, order_num, order_date)
VALUES (7, 'Ink', 5894, '2020-03-10');
SAVEPOINT my_savepoint;
UPDATE Orders SET prod_name='Scanner' WHERE order_id=8;
RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

Handling errors in mysql

- DECLARE action HANDLER FOR condition_value statement;

If a condition whose value matches the `condition_value` , MySQL will execute the `statement` and continue or exit the current code block based on the `action` .

The `action` accepts one of the following values:

- `CONTINUE` : the execution of the enclosing code block (`BEGIN` ... `END`) continues.
- `EXIT` : the execution of the enclosing code block, where the handler is declared, terminates.

The `condition_value` specifies a particular condition or a class of conditions that activate the handler. The `condition_value` accepts one of the following values:

- A MySQL error code.
- A standard `SQLSTATE` value. Or it can be an `SQLWARNING` , `NOTFOUND` or `SQLEXCEPTION` condition, which is shorthand for the class of `SQLSTATE` values. The `NOTFOUND` condition is used for a cursor or `SELECT INTO variable_list` statement.
- A named condition associated with either a MySQL error code or `SQLSTATE` value.

The `statement` could be a simple statement or a compound statement enclosing by the `BEGIN` and `END` keywords.

Examples

```
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
SET hasError = 1;
```

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
    ROLLBACK;
    SELECT 'An error has occurred, operation rollbacked and the stored procedure was terminated';
END;
```

```
DECLARE CONTINUE HANDLER FOR NOT FOUND
SET RowNotFound = 1;
```

```
DECLARE CONTINUE HANDLER FOR 1062
SELECT 'Error, duplicate key occurred';
```

```
CREATE TABLE SupplierProducts (
    supplierId INT,
    productId INT,
    PRIMARY KEY (supplierId , productId)
);
```

```
CREATE PROCEDURE InsertSupplierProduct(
   IN inSupplierId INT,
   IN inProductId INT
)
BEGIN
   -- exit if the duplicate key occurs
   DECLARE EXIT HANDLER FOR 1062
   BEGIN
                SELECT CONCAT('Duplicate key (',inSupplierId,',',inProductId,') occurred') AS message;
   END;
   -- insert a new row into the SupplierProducts
   INSERT INTO SupplierProducts(supplierId,productId)
   VALUES(inSupplierId,inProductId);
   -- return the products supplied by the supplier id
   SELECT COUNT(*)
   FROM SupplierProducts
   WHERE supplierId = inSupplierId;
END$$
DELIMITER ;
```

```
DROP PROCEDURE IF EXISTS InsertSupplierProduct;
DELIMITER $$
CREATE PROCEDURE InsertSupplierProduct(
   IN inSupplierId INT,
   IN inProductId INT
)
BEGIN
   -- exit if the duplicate key occurs
   DECLARE CONTINUE HANDLER FOR 1062
   BEGIN
                SELECT CONCAT('Duplicate key (',inSupplierId,',',inProductId,') occurred') AS message;
   END;
   -- insert a new row into the SupplierProducts
   INSERT INTO SupplierProducts(supplierId,productId)
   VALUES(inSupplierId,inProductId);
   -- return the products supplied by the supplier id
   SELECT COUNT(*)
   FROM SupplierProducts
   WHERE supplierId = inSupplierId;
END$$
DELIMITER ;
```

MySQL handler precedence

In case you have multiple handlers that handle the same error, MySQL will call the most specific handler to handle the error first based on the following rules:

- An error always maps to a MySQL error code because in MySQL it is the most specific.
- An `SQLSTATE` may map to many MySQL error codes, therefore, it is less specific.
- An `SQLEXCPETION` or an `SQLWARNING` is the shorthand for a class of `SQLSTATES` values so it is the most generic.

Based on the handler precedence rules, MySQL error code handler, `SQLSTATE` handler and `SQLEXCEPTION` takes the first, second and third precedence.

Suppose that we have three handlers in the handlers in the stored procedure `insert_article_tags_3` :

Using a named error condition

```
DELIMITER $$

CREATE PROCEDURE TestProc()
BEGIN

    DECLARE EXIT HANDLER FOR 1146
    SELECT 'Please create table abc first' Message;

    SELECT * FROM abc;
END$$

DELIMITER ;
```

```
DROP PROCEDURE IF EXISTS TestProc;

DELIMITER $$

CREATE PROCEDURE TestProc()
BEGIN
   DECLARE TableNotFound CONDITION for 1146 ;

   DECLARE EXIT HANDLER FOR TableNotFound
         SELECT 'Please create table abc first' Message;
   SELECT * FROM abc;
END$$

DELIMITER ;
```

Connect with me on LinkedIn

Please subscribe to our YouTube channel

Check out my GitHub profile

Follow me on Twitter(X)

Thank you